**CSC/ECE 573 (001) - Internet Protocols**
**Fall 2018 Project Proposal**

**An Advanced File Transfer Protocol**
**Efficient & Prioritized Synchronization of Clients File Systems to a Central Remote Server**

---

**Sultan Almutairi - Ramya Challa - Benzil P. John - Michael Lewallen - Wade Moore**

# Outline

1. Introduction
2. Problem Statement
3. Objectives
4. Design
5. Evaluation
6. Demonstration

# Introduction

A greater understanding of the standard network stack and various internet protocols has provided an opportunity for the team of NC State University graduate students listed above to design and develop a solution for cloud storage systems. This solution will allow enterprises to configure any number of client hosts to automatically sync file artifacts to a configured remote server. The team is focused on optimizing network communication between hosts to provide two key benefits. First, to reduce network bandwidth utilization by minimizing redundant file transfers. Second, to allow the prioritization of certain file directories; enabling specific artifacts to be backed up with greater network priority, reducing the risk of potential client data loss during transmission. This proposal will describe how to provide these benefits, the current design of the solution, and how we plan to evaluate and demonstrate the finished product.

## Problem Statement

The majority of data used and stored in various enterprise environments is represented as files. This empowers (Simple) File Transfer Protocols (S)FTP to satisfy data storage requirements even at a large scale. The general scenario this solution pertains to is one where an enterprise requires many clients to synchronize file system data to one or few central servers used for backup storage. Based on one's configuration, the central data could be shared and updated by various clients. However, by default, each client would their own personal storage space reserved. Regardless, enterprise data, which is generated by a multitude of employees or systems, is managed most easily from one or few locations. In addition, we understand business critical data will require higher priority when being synchronized. This solution will backup file data so to avoid any loss in the event of client failures. However, because the data is copied over a network, there will be an inherent latency from the time the data is originally created, and when the data has been completely copied to the remote server. This delay creates a window where the data is vulnerable to being lost. Therefore, artifacts which are deemed more critical will minimize this window by being prioritized during network communication.

# Objectives

- The overall objective is to allow many clients to send new or updated files to a central server. Creating a system where organizations can consolidate data from multiple unix computers, into one or few.

**FTP Client Objectives**

  - User configuration determines the top level directory to “watch”
  - Everything included in this directory will be synced as-is to the central server on a configured periodic basis.
  - Users can provide different top level directories with individual or common ranks.
  - Files sent by the client will be prioritized by order of rank.
  - Files will be read at configured segment sizes, where a checksum is then generated for each segment.
  - The client communicates over HTTP/REST to the server to determine if the file exists, and if so, determines if any segments have the same checksum and position as what is recorded.
  - If the segment already exists, skip.  Otherwise, send.

**FTP Server Objectives**

  - User configuration determines the top level directory to “store”
  - Everything received by clients will be stored in client-specific directories underneath the “store” directory.
  - Maintain a light-weight database containing file, checksum, client, and system information.
  - User configuration allows clients of certain names to have ranks. Clients default to lowest priority.
  - Respond to client HTTP/REST requests to initiate new clients, and respond to file queries based on the current state of database.
  - Accept new connections from known clients for receiving file segments.
  - Receive and apply file segments in order of client rank.

- Both the client and server objectives shall be achieved through Python/Unix applications. User configuration will be applied at startup, then the applications may run autonomously as long as the host system is active.

## Software Design

The design of our system can be simplified into the two separate applications that will be built, the server and client. We anticipate using Python 3.4+ with any applicable packages to perform the HTTP and Socket communication required. Furthermore, Python can perform needed logic and system calls with relative development ease.

The following design will reveal a need for **HTTP and FTP Application layer protocols.** The FTP protocol being our own custom implementation. This implementation will be done through the use of the **Socket programming, for Transport layer file communication.**

**Client Application Source Overview**

*AdvFTP/client/*
| --- *app/*
| | --- *client*
| *lib/*
| | --- *SystemWatcher.py*
| | --- *FileDecoder.py*
| | --- *SendingQueue.py*
| | --- *ClientSocket.py*
| | --- *ClientHttp.py*

The application can be run on the client host by executing the *client* file.

Initial startup will require configuration to enable the client to connect to the central server. The connection is initially made to the server's REST API, where the client posts itself as a new client by its hostname. The server will return a list of port numbers to create socket connections. The number of ports, and in turn the number of parallel connections to a particular client, is determined based on the current load of the server.

After establishing communication with the server, normal execution of the process will entail periodically invoking the *SystemWatcher* to determine if any configured files have been added or changed. The process sleeps for a configured interval if no file changes were made. Otherwise, the file names of interest are passed to the *FileDecoder* so that each one can be processed.

Each file is copied to a temporary location so that the file sent is exactly the same as it was at the time it was selected. This will avoid faulty behavior in the event that the process attempts to sync files that are currently in use. For all segments of each file, the checksum is calculated and temporarily stored. The *FileDecoder* will then utilize the *ClientHttp* module to interact with the

central server. Using the server API to repeat the following procedure for all files in order by rank:

- Determine if a file exists already, if it does not, send all segments (only once)
  * If the directory used is configured for shared access, the response may be that this client must wait until another client is finished syncing. The same 'file existence' endpoint can be used repeatedly to poll availability.
- If the file exists, then the number of segments currently stored for this file is returned.
- Next, determine if a segment's checksum and position matches what the server has currently. If so it can be skipped.
  * This check is done only for segment positions up until the currently stored segment count. Subsequent segments cannot already exist and must be sent.
- If the segment needs to be sent, add it to the *SendingQueue*

The *SendingQueue* will be a separate running thread which loops continuously, each time invoking *ClientSocket* to send the next maximum number of segments in the queue possible. The maximum number of packets to send is determined by the *ClientSocket*, which will only manage so many open connections at once. This maximum really derives from the server which will only allow so many open connections. The use of these multiple connections allows sending multiple file segments in parallel. This logic will be encapsulated by the *ClientSocket* module.

**Server Application Source Overview**

*AdvFTP/server/*
| --- *app/*
| | --- *server*
| *lib/*
| | --- *Server.py*
| | --- *ReceivingQueue.py*
| | --- *SystemDB.py*
| | --- *ServerSocket.py*
| | --- *ServerHttp.py*

The application can be run on the server host by executing the *server* file.

Initial startup of the server application will entail loading user configuration, and then instantiating a SQLite system database. SQLite provides a lightweight solution to store information using Python. The database is created as a single file in a specified location which holds all schema data. Thus far, the following schema is required:

*Clients* - client_name | client_directory

*Files* - file_name | file_location | clients_with_access | is_active
*FileSegments* - file_name | segment_position | checksum
*Transactions* - tid | file_name | size | total_delay | net_delay | io_delay

The server's main function will be to act as a REST API listening for client communication. The server will have defined endpoints for client provisioning, file existence, and checksum comparisons. Using either Python's Flask or Connexion module the *server* will define these endpoints.

The *ReceivingQueue* will define logic for creating two separate threads to looping continuously. One thread will invoke the *ServerSocket* to read segments from all open connections, while the other applies newly read segments to the local file system. During each iteration, the segments read are ordered by client rank and then added to this "writing queue." The *ReceivingQueue* module maintains these two separate queues; one for accepting new segments over the network, and then another for applying new segments to system files. Writing segments to a file will require locks before being applied.

**Conclusion**

The system requires both Unix machines are able to communicate freely over a network, and that the host information to do so is provided as configuration to both client and server. Subsequently, the applications may operate together autonomously reflecting file system changes as they are made over time.

# Evaluation

To evaluate our project, we will use additional updates to the database to record performance data. Although this will pose a latency, our team hopes the lightweight DB solution is efficient. This cost is accepted being the additional database entries will provide a significant benefit to our research. The table will contain the amount of delay, in seconds/milliseconds, of every client transaction. A transaction consists of collecting *n* segments of a single file from a client, and then updating all *n* segments in that file locally. Provided this definition, the CPU times recorded in the *Transactions* table, is the amount of time taken to receive the entire file (network delay), to update the local file (i/o delay), and the amount of time total (total delay).

The team will strive to develop a simple test script which will create files at various sizes, and at various locations within a directory. The script will mimic the spontaneous nature of production use, and will perform varying degrees of stress on the system. This would provide the ability to aggregate large amounts of data through our test scripts and *Transactions* table, where the team can provide statistical metrics. However, project priorities may only allow for manually file manipulation to be used in analysis. Regardless, for all configuration options as inputs, the team will determine the average, standard deviation, and range of the network, i/o, and total

delays. Various inputs would be the segment size, file size, and/or the number of parallel connections allowed. The average network delay, i/o delay, and total delay can be shown across different variations. Specifically key variations which exploit comparisons of interest, including a very large segment size, defeating the purpose of segmenting or no parallel connections to enforce synchronous updates.

# Demonstration

The demonstration will be graded out of one hundred points. Our demonstration provides four key performance indicators (KPIs) which will each be granted a portion of the hundred point total. The demonstration itself will need to involve several steps to prepare and execute.

**Demonstration Procedure**

- Assemble the appropriate equipment needed to provide a real-time visual depiction of our running applications. Assemble at least two separate machines (laptops, etc) which are connected to a common network.
- One will run the server application.
- All other machines will run the client application, with a team member there to create test files. Only one client is required to perform the demonstration; however, more could be used in a more comprehensive presentation.

**Demonstration Display**

- This would likely be at least two monitors/projections.
  - One display could show user terminal sessions for both server and client in the configured directories so to show and manipulate files.
  - Another display could also show the System Database Transactions table on the server machine. This would continuously refresh so everyone can see when files have finished syncing along with the times recorded.
  - A final single display could show the server and client(s) stream of logs.

**Key Performance Indicators**

KPIs will be verified through the existence and order of transactions added to the system database. Furthermore, our team can compare file checksum values against server and client file systems. The weights corresponding to the the one hundred point grading scale reflect the goals of our solution stated in the problem statement.

- ***Autonomy (10%) -*** Both server and client applications should run autonomously on separate Unix machines until manually stopped. Verification is obvious, and will serve as a base check of system functionality
- ***File Integrity (30%) -*** Verified via checksum comparisons, the applications should effectively send client files to the server file system, and update them periodically upon change. Being files are sent as segments, this implies files are constructed and patched correctly.
- ***File Priority (30%) -*** Through proof of seeing the transaction table order. Transactions initiated on the same sync iteration, and of the same file size, should finish in order of priority.
- ***User Multiplicity (30%)-*** The server file system should reflect multiple directories for each user supported, and recognize each one upon reconnecting.